SLEUTH

THE ULTIMATE GUIDE
TO GOING FROM

ZERO TO ONE HUNDRED DEPLOYS A DAY

Table of contents

How elite teams approach software delivery

The best software teams at top companies like Amazon, Google, Netflix, and Atlassian today are deploying to production hundreds of times a day. Why? They've learned that to deliver world class value to users, they need to be able to iterate on products quickly, ship bug fixes in hours, not weeks, and do so while reducing the risk of unintended downtime.

But it isn't just the software giants who have changed how they work. Software development and delivery is undergoing a radical transformation driven by widespread adoption of DevOps.[1] The backbone of this shift is Continuous Delivery[2], or the ability to deploy frequently. This change began well over a decade ago and has moved into the mainstream in the last five years. Teams with elite DevOps practices have more than tripled in the last year alone.[3]

If you're in an organization that's still struggling to deploy once every six months, you might be thinking, "this kind of thing isn't for me." Often, teams that haven't embraced Continuous Delivery worry that frequent deploys will rob them of control over their release process, cause a dramatic decline in software quality, and take years to adopt. Research shows[4] — and we'll explain how — the reality is exactly opposite.

It can be hard to change what you've always been doing and understand what kind of investment it would take to make the change. I assure you Continuous Delivery is for your team. In this book we will show you why it will help you, how to make your way along this journey, and point out common pitfalls and how to avoid them.

The value of adopting frequent deploys

You may be asking yourself: Why would I want to make the effort to change how we've always worked in the past? The answer is simple: the benefits of Continuous Delivery and frequent deploys are too enormous to ignore.

The biggest benefit is the increased speed, quality, and reliability of your software delivery. Studies have shown that high performers of DevOps practices deploy 46x more frequently, are 440x faster at getting committed code to deploy, are 170x faster to recover from downtime, and have a failure rate 5x lower than lesser performers.[5]

Studies have also found that DevOps practices correlate strongly with job satisfaction. Not only will your teams deliver higher quality and more reliable software quicker, but they'll be happier doing so.

From the speed, quality, and reliability improvements comes increased business value. Your organization will be able to fix bugs and security issues in hours instead of weeks, deliver new customer-facing value at a fast pace, and respond to customer feedback in near real-time.

These capabilities also come with more autonomous teams. Autonomy means you can more easily scale your organization, empowering smaller teams to focus and ship their area of focus independently of other teams inside your organization.

Finally, because your team will be working and measuring in close to

real-time, your organization will become nimbler. You'll be able to detect and react to mistakes in direction or execution quickly.

Another benefit that can't be overstated is the overall impact on your organization's culture. To fully adopt Continuous Delivery and frequent deploys, you'll have to have a culture of continuous learning. This implies a blameless culture of software development.[6]

A culture of learning, continuous improvement, and of personal and team responsibility is a strong culture, and organizations with such culture do a better job attracting and retaining the best talent.

What it looks and feels like to deploy many times a day

Every team will look somewhat different, but when you're at the start of a journey it's exciting to have an idea of what your destination can look like. Here are some of the exciting things you'll be capable of:

- ✓ PM and Design work right alongside developers, iterating as you go, often conceiving of and shipping work within a day or less!
- ✓ A customer suggests a small improvement or points out a small bug and you're able to respond hours later to let them know it's fixed. It's a conversation instead of a one-way interaction.
- ✓ Your test suites and deployment pipeline mean you deploy with safety and confidence. Your deployments are a non-event.
- ✓ You can instantly turn on and off features in production through feature flags.

- ✓ You have multiple environments in place as safety gates so that you rarely negatively impact production. And, if you do, you're able to roll back in minutes, not hours!
- ✓ Code "working in production" is a natural part of a developer's definition of done
- ✓ Your developers truly own their changes and are alerted in Slack when they've made a change that had unintended consequences
- ✓ You have the flexibility to differentiate a simple, fast-tracked change from a riskier change that requires approval from the right people in your organization
- ✓ Everyone in your organization knows, via automation, what's released when they need to know it
- ✓ You have accountability, visibility, and confidence to make changes multiple times a day!

Getting started on the journey

If you're on a team that's already on your DevOps journey, deploying once a week or even daily, we will show you why you want to keep improving and what hurdles await you in the next step of your journey. We'll explain what elite teams, at scale, are doing today to overcome the challenges you'll soon face.

We've identified three phases of Continuous Delivery adoption:

1. **Deploy once a week**
2. **Deploy once a day**
3. **Deploy a hundred times a day**

Each phase comes with four categories of milestones:

- **Measurements** – Tracking metrics and setting goals. By metrics I'm specifically referring to Accelerate metrics (also known as DORA)
- **Development practices** – Implementing practices and tooling to support frequent delivery efforts, including CI/CD[7]
- **Communications** – Creating communication lines
- **Culture** – Developing mindsets, rituals, and accountability

In this book we will walk you through each phase in detail. For each phase, we'll identify the key milestones and the tooling to support your team's progress in its Continuous Delivery journey.

01

Measuring where you are today

STORY FROM THE TRENCHES

Flying blind and a Sleuth is born

When I started running the Bitbucket team in 2010 we were early adopters of DevOps and frequent deploys, deploying about two or three times a week. We were a small team, growing fast, and maturing our process along the way.

A couple of years later we had a staging environment in place, 500,000 users, a team of about 15 developers, and we were deploying between 10 - 20 times a week. I started to get questions from management about how our software delivery was going. At the same time, we started to see more incidents and instances of developers stepping on each other's toes. It became clear we were going to struggle to improve with the visibility we had in place.

That's when I realized we were practicing DevOps but that we were flying completely blind! Even though we were building a premiere source code hosting and code review tool and we worked at the Jira company, we couldn't tell what was actually changing in each deploy. We had almost no insight into how long it took us to take things from code to production. We had very little insight to how our changes affected production performance over time. Worst of all, every incident was a fire drill where we struggled to link up eight different systems to quickly figure out what was going wrong. We built a few home-grown tools to mitigate the worst of these impacts. However, I was struck by how deploys — the core of what we do and the real moment when the rubber hits the road and your code ships to customers — was treated as an afterthought with very little instrumentation.

When I left Atlassian in 2014, I started to fiddle with the beginnings of what would become Sleuth. The idea was simple: deploys are the most important thing, so let's track them and make it dead simple to see exactly what changed and how to get to those eight different systems for each deploy. Link up observability tools so we can see when a deploy has made something worse and how long it takes to get better. Finally, build a robust version of the homegrown tools to bring order to the world of frequent deploys.

My big takeaway from the experience was you can't improve what you don't measure. And you can't improve with a large team, at scale, without world-class software tools to help you.

You can't improve what you don't measure

Whether you've yet to begin your journey to weekly deploys or you're already deploying many times a day, your first step is to understand where you're at today.

It's important to remember that DevOps, like Agile, is a philosophy. There is no one right way to achieve the goal of more frequent deploys. There are some things, like automated builds and CD pipelines, that all teams will adopt.

But remember, the right strategy for your team will be specific to your team. If there's a tool, process or cultural hurdle your team can't overcome, that doesn't mean you're stuck. You can always find the unique mix that will work for your team.

ESTABLISHING YOUR TEAM'S BASELINE IS A FOUR-STEP PROCESS:

1. Baseline your team's Accelerate metrics[8]
2. Baseline the technical tooling your team has in place
3. Baseline how your team communicates about changes
4. Baseline your team's cultural attitude toward risk, frequent changes, and what's most important when delivering software

In each section below, you can run through our checklists and score your team as: Don't have (0 points), Novice usage (1 point), Intermediate usage (2), or Advanced usage (3).

If you're not sure what constitutes a level of usage, simply use your best guess, as this is just a rough baseline.

Measuring DORA / Accelerate metrics

One of the most popular topics to come out of the book "Accelerate," are the "four measures of software delivery performance," sometimes called "Accelerate metrics" or "DORA metrics."

These metrics are:

- ✓ **Deploy Frequency** — how often your team deploys to production
- ✓ **Change Lead Time** — how long it takes for a change to go from first code commit to deployed
- ✓ **Change Failure Rate** — the percentage of deploys deemed a "failure"[9]
- ✓ **MTTR or Mean Time to Recovery** — the amount of time it takes to recover from a "failure"

Extensive research has shown that these measures correlate and have a direct impact on how often a team is able to deploy and with what level of confidence. Measurements allow us to divide teams into four groups: Elite, High, Medium, and Low performers.

The table below shows how these different groups stack up for each metric:

	ELITE	HIGH	MEDIUM	LOW
Deploy Frequency	Multiple times per day	Between once a day and once a week	Between once a week and once a month	Between once a month and once every 6 months
Change Lead Time	< 1 day	Between 1 day and 1 week	Between 1 week and 1 month	Between 1 month and 6 months
Change Failure Rate	0 to 15%	0 to 15%	0 to 15%	46 to 60%
MTTR	< 1 hour	< 1 day	<1 day	Between 1 week and 1 month

There are a number of open source and commercially available tools that help you measure your Accelerate metrics. Most of these tools take the approach of inspecting your Git repositories metadata to make inferences about the metrics.

A better approach would be to gather data from not just version control systems, but also build, monitoring, and alerting or incident management tools, so you can get a very clear and accurate picture of these metrics. The extra accuracy you get is worth the extra effort.

The best tools will also call out outliers within these metrics and provide actionable suggestions on how your team could improve.

No matter how you decide to measure, you'll want to understand where you are at and have an ongoing way of seeing these values as you try to move your teams forward. In later chapters we'll go into more detail about how you can improve in each metric.

Measuring your technical tooling

There are innumerable tools in the development ecosystem, but we can group many of them into broad categories, each of which can be key to the adoption of Continuous Delivery. Having the tooling in place is your first step.

HOW IS YOUR TEAM USING THE FOLLOWING TOOLS?

- **Source control** (e.g., Github, Gitlab, Bitbucket)
- **Code reviews** (e.g., pull requests, merge requests, code walkthroughs)
- **Automated testing – CI** (e.g., Jenkins, CircleCI, GitHub Actions)
- **Deployments – CD** (e.g., AWS CodeDeploy, Harness, Jenkins)
- **Observability** (e.g., Datadog, New Relic, CloudWatch)
- **Error tracking** (e.g., Sentry, Rollbar, Bugsnag)
- **Feature flagging** (e.g., LaunchDarkly, Split, Cloudbees)
- **Deployment environments** (e.g., Production, Staging, QA)
- **Configuration as code** (e.g., Terraform, Chef, Puppet)

Measuring your communication lines and tools

The key to moving fast is having open and well-defined lines of communication, both within and between your teams.

This can be as broad as an executive knowing if a high-level feature has shipped, or as narrow as a developer knowing when to merge code to be included in a deploy. As you increase the number of deploys your team performs, your lines of communication become more and more important.

Here we are measuring both the tools that help you communicate (and the maturity of your use) and the lines of communication that exist.

HOW IS YOUR TEAM USING THESE COMMUNICATION TOOLS?

- **Issue tracking** (e.g. Jira, Pivotal tracker, Linear)
- **Chat tools** (e.g. Slack, Teams, Discord)
- **Visibility – deployment tracking** (e.g. Sleuth, Harness)
- **Knowledge sharing** (e.g. Google Docs, Confluence, Notion)

DEVELOPER COMMUNICATION

- How often do your developers work together? Through what means? Slack? Issue tickets? Pair programming? Code reviews?
- How do you communicate processes, like deploys, to your team?
- How does your team improve? What processes exist for this?

OPERATIONAL COMMUNICATION

- How do you understand the health of your deployment environments? Is this available to all or just an anointed team? How hard is it to find the right information?

INTRA-TEAM COMMUNICATION

- How do your developers communicate with external stakeholders such as product managers and designers? How often and how directly? Hourly, daily, weekly, monthly? Only through a proxy?

EXTRA-TEAM COMMUNICATION

- How does the rest of the company know what your team has accomplished?
- How do others know what your goals are and if you're hitting them?
- How do other functions that rely on your work know when it's done?

Measuring cultural attitude toward frequent deploys

One of the biggest challenges to adopting frequent deployments isn't the technical or communications challenges, but the cultural shifts every team in the organization needs to make and buy into.

Culture is the hardest thing to shift, but there are tried and true rituals that can shift even the most dug-in of teams. However, it does require an open mind, and the willingness to learn and, at times, fail to eventually succeed.

The cultural hurdles you'll face can be grouped into four broad categories. Where does your team stand with:

CULTURE OF SAFETY

- How does your team verify production is in a healthy state?
- How blameless is your team when you encounter incidents?
- Are your developers supported in creating the tests and environments that allow for them to safely make changes?

CULTURE OF CONFIDENCE

- How do you gain confidence in the changes you release?
- Do product managers, engineering managers, and design and engineering teams feel like they're on the same team? Do they trust each other's work? How do they verify this?
- Does your team know how to ship and verify incrementally and "incomplete" changes or does it only know how to do this at the feature level?

CULTURE OF OPENNESS

- How does your team communicate about what's changing?
- How does your team define success? Is this measure open to the team, the execs, and the entire company?
- Is it ok for your team to fail and learn? How does your team support this?

CULTURE OF RESPONSIBILITY

- Do individuals own their work?
- Do your developers own their code? At what point are they no longer able to see a change through? (dev, testing, review, merge, staging, prod)
- Do individuals feel empowered to improve the processes they work within?
- When things break is it our problem or someone else's problem?
- Who owns reliability?

Generating your baseline

Once you've started tracking your Accelerate metrics, you can bucket yourself into one of the four categories: low, medium, high or elite. This provides a clearer picture of what your next goal should be.

Once you've gone through the other sections and scored your processes, you can average your scores across each section. This will result in a score between 0 and 3 for technical tooling, communications, and cultural attitudes.

Understanding where your team is strong or weak will give you a good idea of where to focus first to start making progress.

In the next chapters we'll talk about how development practices, communications, and culture can be changed to help drive improvements on your Accelerate metrics. In this way — measuring, identifying a small improvement, implementing, and re-measuring — your team can learn to deploy a hundred times a day.

02

Deploy once a week

STORY FROM THE TRENCHES

Reciprocity's journey to a deploy a week

When I joined Reciprocity as a Chief Architect, the small company of about 40 developers was deploying new releases every three months. Releases were high-risk affairs as the development team was surprisingly productive, and so releases contained a huge amount of changes.

For example, during one of my interviews, I sat on the sidelines and watched as a release seemed to introduce a security issue, causing them to shut down the service completely. It turned out there was no issue but a poorly understood access mode that an employee stumbled upon, but the reaction to the incident was interesting.

Rather than review these high-risk, big-bang deployments, the inclination was to slow down and add more manual testing in a futile attempt to "get it right." I've seen this reaction again and again from small to big teams, not realizing that by slowing down, the risk actually increases due to the larger number of changes.

In the first six months, I put them on a path to deploy code to production once a week, with an occasional daily deployment of a small, isolated change. Automating the release was the first step, but I quickly discovered the real challenges weren't technical.

The entire company was built on infrequent releases. Marketing was set up to only promote new features quarterly. Sales, while loving that we could respond to requests much quicker, was so frustrated with the frequent changes that we had to build a separate cluster that was updated separately.

Quality Assurance had the biggest adjustment, but surprisingly, turned out to be our biggest advocates. Recognizing we had to rethink how we tested changes so that releases wouldn't be blocked, we set aside engineering resources to help them automate and modernize their testing automation stack. This was something they'd been wanting for months, so they were thrilled to get it now.

My big takeaway from this experience was to remember it affected people across the whole company and not just a code base. People's workflows, tools, and even quarterly goals were affected, so you need to identify friction points ahead of time and devote energy to bringing them along and ensuring their lives are clearly better for it.

When you get to Phase 1 where you can easily deploy once a week, you've officially crossed the threshold into the world of DevOps and continuous delivery. The muscles your team develops to get here are the foundation upon which further strides will be made.

The keys to your team being able to deploy once a week are:

- Tracking of Deploy Frequency and Change Lead Time
- Well-defined source control, code review and reproducible CI builds
- A mechanism for intra-team communication so they know what and when changes will be deployed
- The cultural rituals and support that give the team confidence that it's OK to deploy and to handle things if a deploy goes poorly
- A clear "owner" for deploys. Everyone on the team needs to know how they fit within the process and who to look to if they have questions or concerns.

Measurements for one deploy a week

Measurements are critical for knowing your starting point and current progress. You'll want to have at least the tracking in place for two Accelerate metrics: Deploy Frequency and Change Lead Time.

Note that Change Failure Rate and MTTR, the other two of the big four Accelerate metrics, are certainly helpful at this stage but become much more important when you start moving to daily deploys.

To achieve a Deploy Frequency of once per week, your goal is to drive Change Lead Time to at or below five days.

You can achieve this by reducing your batch size. Smaller batch sizes means you are making smaller changes, which implies less risk. It also means that you can develop a “fix” between deploys if need be.

You’ll need tooling that provides at least the following capabilities:

- Track and compare Deploy Frequency and Change Lead Time trends over customizable periods.
- Drill down on Deploy Frequency and Change Lead Time by codebases and flags.
- Drill into Change Lead Time to see where time is being spent, such as time spent on coding, code reviews, and deploys.
- Identify outliers in deploys.

The ideal, next-level capabilities you should look for include:

- Slack-based team deploy notifications to provide visibility when changes are shipped.
- Slack-based personal deploy notifications to authors of changes so they can instantly know when things ship and own their changes.
- Automated checks against batch size, triggering either a notification or an approval process when it finds the batch size is too large.

Development practices for one deploy a week

In terms of development practices, your goal is to have a mostly repeatable deployment process that doesn't take longer than four hours for an individual to perform. You'll also want to have some mechanism in place to evaluate the success of deployments. Finally, you'll need the ability to revert your changes in a timely manner if the need arises.

Adopting the following practices will help you achieve them.

SOURCE CONTROL

- ✓ You're using developer branches for new changes
- ✓ You maintain a "release" branch that is always in a deployable state. Developer branches are merged into this branch.

CODE REVIEWS

- ✓ You have a mechanism for accepting or rejecting changes into your release branch
- ✓ Ideally, you are using pull requests[10] to review new changes, and you have a well-defined process for merging these changes

TESTING

- ✓ You have at least 50% unit test coverage
- ✓ Your unit tests are at most 20% flakey[11]
- ✓ Your unit tests run on every code commit

DEPLOYMENTS

- ✓ You have a mostly automated continuous delivery pipeline in place that can build a “release” from your “release” branch
- ✓ You have a well-defined production environment and a strong understanding of how changes affect this environment

OBSERVABILITY

- ✓ You have basic observability in place, such that you can judge the success of your changes to production

Communications for one deploy a week

In terms of communications, the goal is to have the lines of communication and tooling in place such that developers and PMs know what’s going into a deploy and, more important, the process by which the deploy happens.

Developers need to know how to get their changes included in a deploy, and how to communicate the risk of those changes to the team. PMs need to understand when changes will deploy so they can verify work and communicate changes to customers.

The following practices will help you achieve the goal.

DEVELOPER COMMUNICATION

- ✓ You have a well-defined day/time for when changes need to be integrated into your release branch such that they will ship with the weekly deploy.

- ✓ You have a well-defined window of when the changes will be live in production.
- ✓ You have a well-defined understanding of who is responsible for determining if a deploy is successful. This can be a build engineer, automated alerts, or the developers who made the changes. The important thing is to know who makes the call to rollback or keep your changes live.

INTRA-TEAM COMMUNICATION

- ✓ You have a way for product managers, engineering managers, and developers to know when a deploy has completed and what was included.

ISSUE TRACKING

- ✓ You are using an issue tracker to, at a minimum, track the high-level work that's going into a deploy. This provides the basic language for communicating development work between developers and to other team members such as product managers and designers.

REAL-TIME CHAT

- ✓ You have a mechanism for developers and those who own the deploy to have real-time communication (e.g., via Slack).

VISIBILITY

- ✓ You have a way to know, even if it's cumbersome, what code and high-level tasks have been deployed.

KNOWLEDGE SHARING

- ✓ You have some form of documentation that describes how your deployment process works, who owns it, and who is a point of contact

to find out more. Ideally your process is self-documenting, but that's often unrealistic, even when you deploy many times a day.

Culture for one deploy a week

The goal here is to provide the entire team with the confidence required to make and support rapid change. If you are moving from a monthly or quarterly deployment cycle, this will likely be the largest challenge in getting to one deploy a week. A motivated engineer or leader can put in place the required technical tools and communication lines. However, it requires your full team's buy-in to commit to a weekly deploy cadence.

Let's understand the cultural buy-ins and some of the means to achieve them so each role on your team can make this shift a success.

DEVELOPERS

Developers need to understand and buy into the team's dev loop. They need to be trying to make smaller changes and target their changes to deploy windows. They need to agree that a break in the release branch trumps other concerns and will be fixed immediately.

Mechanisms to achieve:

- ✓ Weekly or bi-weekly sprint planning can help set the size of tasks and create a team agreement on how tasks will fit into this window
- ✓ Automated CI against the release branch and team-wide visibility into those results to help the team keep the release buildable
- ✓ Maintaining a "disturbed" role on a team where it's clear whose job it is to keep the release branch buildable

PRODUCT MANAGERS AND DESIGNERS

PMs and designers need to change their idea of what it means to ship functionality to customers. They need to work closely with Dev to set expectations of when something customer facing can be exposed and be flexible about how incremental changes make its way to production.

Design can still design the end goal of how a thing should operate; however, they may need to spend more time and effort on intermediate designs that can be shipped incrementally along the way.

Mechanisms to achieve:

- ✓ Include a PM in the planning process to give them an opportunity to express the needs of the customer in incremental deploys.
- ✓ Implement feature flags, as it allows for code to be rolled out but not yet exposed to customers. This helps PMs retain the flexibility they need while allowing developers to charge ahead.

ENGINEERING MANAGERS

Engineering managers need to shift from putting in processes that help the team play defense (e.g., "How can we add processes to slow down and make sure our 3-month release has no bugs and is 'safe' to ship?"), to putting in processes that help the team play offense (e.g., "How do we keep the code flowing? What can we do to ship smaller changes and quick fixes if we make a mistake?").

Mechanisms to achieve:

- ✓ Put in place processes that facilitate communication such as recurring planning meetings, code review, and weekly demos.
- ✓ Commit to dedicating engineering time to keep the release branch builds passing. Keeping the code flowing also means paying into this

new system. When the team identifies a bottleneck, the manager should be able to allow the team to remove it (e.g., fix CI tests that have a flakeyness higher than 20%).

UPPER MANAGEMENT

Before frequent deploys, upper management could evaluate the team on the release boundaries: "If releasing quarterly, how did it go? Did they hit objectives?"

With one deploy a week, these high-level measures can remain in place, but things like MTTR and overall application quality come to the fore-front as leading indicators of success or failure.

Management should expect a team's overall performance to increase by adopting continuous delivery, but the measures and timeframes need to change.

Mechanisms to achieve:

- ✓ Surface the Accelerate metrics for your projects and make them available, with context, to your execs.
- ✓ Surface your uptime or equivalent for your applications.
- ✓ Make sure to provide your exec team with the same high-level information about large chunks of functionality that are shipping.
- ✓ If you are an exec, understand that change takes time and be firm about holding your teams accountable for high-level goals but flexible about how they achieve them. Trust, but verify.

Who owns your one deploy a week?

Any successful process requires you to clearly define who owns it. This allows your team to understand where to look for leadership and guidance and makes it clear who, or which team, is your main point of contact for issues or questions that arise.

When your team is deploying multiple times a day, you'll find that developers should own their own process for deploying their changes. However, when deploying once a week, you have more options.

OPTION 1:
SRE OR INFRASTRUCTURE-MINDED DEVELOPER OWNS THE DEPLOY (RECOMMENDED)

If your team is lucky enough to have one or more team members that understand the code and the operational environment it lives in, these folks can be a great choice to own your weekly deploy.

The upside to this is that you're moving dev and deploy closer together. Most issues that arise from frequent deploys are code-related, so having someone that understands this and has a close relationship with your developers can really help to break down the boundaries between Dev and Ops.

OPTION 2:
BUILD TEAM OWNS THE DEPLOY

Many organizations transitioning to frequent deploys have a whole

team in place whose responsibility is to release the application. This team can play a role in transitioning to frequent deploys.

The upside to having this team own the deploy is that they likely are most familiar with your build and deployment pipeline and they already are used to evaluating the health of your production environment. The downside is that this team may not be fully open to the transformations required to really make frequent deploys a success.

OPTION 3:
DEVELOPERS OWN THE DEPLOY

This might seem like your best option since you'll want this to be the case when you're deploying daily. However, you may be jumping the gun here.

Developers don't always have the zoomed-out context to understand all the changes that are going into a deploy. Because you are bulking up a week's worth of change into one deploy, you will want someone who really understands the system as a whole to be at the helm. If this is your dev team, awesome!

However, if you are transitioning, be mindful about putting too much responsibility onto your developers' shoulders without providing them the support and tools needed to be successful.

03

Deploy once a day

STORY FROM THE TRENCHES

Statuspage's journey to a deploy a day

When I joined the Statuspage team as Head of Engineering, the team was still a founder-led engineering team. The CEO was the only one able to deploy even though there was a team of five contributing to the codebase. Within three months, we were deploying one to five times a day and all engineers were empowered to deploy their own changes.

The founder who was holding onto deploys had already done the work required to get to one deploy a week. The biggest barrier to moving faster was the founder's worry that we wouldn't be able to mitigate the risk that comes with frequent deploys.

There were also some basic tooling and practices missing that hindered us:

- The team wasn't using a well-defined mechanism for merging code changes into our release branch.
- Our batch sizes were very large, increasing the risk of each deploy.
- The team wasn't doing planning, so we had no idea when work would ship.
- We had manual steps in the deploy that weren't documented or well defined.

I knew we couldn't grow this way and decided this would be my first large organizational transformation with my new team. The first step was building awareness and some buy-in for the idea of change. We started by introducing pull requests and light-weight code review. This provided a way for every developer to consistently signal intent to make a deploy. Next we introduced a short planning meeting, giving us our first bit of developer cadence. We could start to think in terms of batches.

The founder was happy with our initial strides, but he was still quite skeptical that we could parlay that into everyone being able to safely deploy. The next step was to convince him that others could handle emergency situations if they arose. We added myself and another to our ops rotation. At the same time, we worked to make our deploy process repeatable and not something one did from a laptop.

The journey to a fully automated process and a CD pipeline gave the founder time to see that change was possible without compromising safety. He also began to feel the relief of not being the only one who could deploy and of being the bottleneck for his growing dev team.

With these changes in place, the final change of letting everyone deploy their own changes showed up with a whimper, not a bang. In the end, this transformation was 75% cultural and only 25% technical.

When you can easily deploy once a day, you've unlocked a whole new way of working. You're now able to fix bugs within hours, not days. Your developers can deploy large changes one small piece at a time, reducing risk by verifying changes as you go, instead of in one big bang. You can be truly agile and iterate as you build. Most important, your team's velocity will no longer be held back by their ability to get changes into your systems.

The keys to your team being able to deploy once a day are:

- Measuring all four Accelerate metrics: Deployment Frequency, Change Lead Time, Change Failure Rate and MTTR.
- A fully automated CI/CD pipeline that can reliably deploy changes to your multiple environments and the ability to quickly revert.
- Observability and alerting capabilities that allow you to quickly detect when changes have caused negative impact.
- Cultural rituals and support that provide the team with the ability to self-serve their deploys, the confidence to understand what to do if things go wrong and, for non-technical team members, how to know what's changing.
- A strong buy-in from the whole team around working in small batch sizes and shipping features in a continuous fashion.

Measurements for one deploy a day

To support one deploy a day, you will want to have measurements in place for all four Accelerate metrics. For a deploy a week, you've

focused on frequency, batch size and its related lead time. Moving to a deploy a day, your focus shifts to reliability and the safety net in place to move at speed.

Your primary goal to achieve a Deploy Frequency of one deploy a day is to have a trustworthy measurement of your team's Change Failure Rate and its associated MTTR.

Because you're moving at a faster pace, your team's Change Lead Time should be at or below two days. This means you are reducing your batch size such that each change is somewhat self-contained and can easily be reverted.

Teams that are classified as Elite or High performers (those who deploy once to multiple times a day) find that they need to keep their Change Failure Rate under 15%. Anything above this failure rate means you're spending all your time rolling back changes and not actually delivering value to your customers.

That said, there's a lot of interpretation and nuance involved in defining what "change failure" means for your team. On one end of the spectrum, failure can be defined as an incident where your service or product is hard-down. One the other end, it could be as sophisticated as your product violating an internally set service level objective (SLO).[12]

Examples of a sophisticated definition of change failure are:

- Error rates breaching your team's norms
- Response times exceeding a threshold
- Emails not arriving in customers' inboxes within a set timeframe

It's important for your team to think about how to define change failure and to use tools that allow you to accurately measure this. Teams that are deploying daily want to keep their MTTR to less than one day and ideally to less than one hour.

Your tooling should be able to report, at a minimum:

- Change failure, broken down by different code bases and flags you are changing
- Quick access to outliers related to your failure rate and deploys
- A system that tracks what's normal for your team and calls out when you're outside that norm
- The ability to quickly drill into the original source of measure (e.g., Datadog)

The best tooling your teams can employ to improve toward your goal of one deploy a day are:

- Slack-based notifications at the team and individual level that identify when you've encountered change failure, which can drive your mean time to detect (MTTD)[13] to zero
- Slack-based approvals when promoting from a non-production environment to your production environment
- Deployment locking, allowing your team to throw on the brakes when change failure has occurred, so you don't pile changes on top of a broken system
- Enforced and automated soak time for pre-production environments
- An automated system that allows smoke tests to stop production promotions

Development practices for one deploy a day

The goal for technical tooling to support one deploy a week is to have a fully automated process that 1) doesn't take longer than two hours to complete, and 2) is triggered and executed in a CD pipeline.[14]

You also want to have at least one alternative pre-production environment, so you can try out riskier changes there first.

Finally, moving at this pace means you need more automated observability into how your systems are performing. When you're deploying daily, it would become a full-time job for an individual to verify every change, so this must be passed off to automation.

The following practices will help you achieve this.

SOURCE CONTROL

- ✓ You're using pull requests for your changes and able to create revert pull requests if needed.

CODE REVIEWS

- ✓ You have a process that your team has committed to, such as PRGB (pull request review, green build). Most popular code review tools allow you to enforce a minimum number of reviewers before a change can be merged.

TESTING

- ✓ You have at least 75% unit test coverage
- ✓ Your unit tests are at most 10% flakey

- ✓ Your unit tests gate your deploys
- ✓ You have end-to-end tests covering your application's critical paths
- ✓ Your end-to-end tests run against an environment that is a close approximation to your production environment

DEPLOYMENT ENVIRONMENTS

- ✓ You have at least one well defined pre-production environment that closely mimics your production environment

OBSERVABILITY

- ✓ You have advanced observability in place with automated alerts
- ✓ You have a well defined on-call roster with a clear escalation chain defined
- ✓ Your deployers have access to observability metrics and can diagnose production systems

ERROR TRACKING

- ✓ You have a way, either through log analysis or an error tracking tool, to understand the errors your system produces. You should be able to quickly discover if a change has increased the number of errors.

FEATURE FLAGGING

- ✓ You should be able to, at a minimum, hide certain features and code paths from your customers so you can deploy your changes incrementally, without having to expose your incomplete work to your customers.

CONFIGURATION AS CODE

- ✓ Once you're maintaining multiple environments you need to have

a way to make sure that infrastructure changes flow to all of them. Keeping these systems in parity can be difficult without some form of configuration as code.

Communications for one deploy a day

The goal here is to have the lines of communication and tooling in place, such that the team has a strong understanding of the cadence of activities, knows how to communicate to others about deploy related issues, and understands what to communicate when things go wrong and how to get them back to normal.

Adopting the following practices will help you achieve this goal.

DEVELOPER COMMUNICATION

- ✓ All developers know how to trigger your deployment process and are comfortable with executing it. Because they happen frequently, you'll need some shared channel to disseminate to the team when a deploy is happening and what's included.
- ✓ You have a way for everyone to see, understand, and communicate the important health metrics for your system. This can be a shared metrics dashboard, a shared Slack channel for alerts, or impact tracking in Sleuth.
- ✓ You have a well-defined and automated escalation policy for when deploys go wrong. Even in the best teams, unexpected things go wrong. To move with confidence, individuals need to know they're not alone, and they need to know they can easily call in the cavalry. Tools like PagerDuty are great for this.

- ✓ Your team has a well-defined cadence for how they produce and ship their work. Your team needs some planning mechanism that allows them to define work in batches small enough to fit into a daily deploy cadence.

INTRA-TEAM COMMUNICATION

- ✓ You have a way for PMs and EMs to understand the current overall progress toward a larger goal (e.g., an epic).

When you start deploying in very small batches, this can become one of your biggest challenges. What used to be a binary answer (it shipped, or it didn't) has now become much more complicated (how far along is it and when will it be ready to be visible to customers). Project management tools or tools that associate deploys with epics along with notifications can help teams stay in sync.

OPERATIONAL COMMUNICATION

- ✓ You have a clear channel of communication between developers and those responsible for your systems infrastructure (if these are the same people in your team, you still want well-defined roles and communication channels).

Deploying daily requires confidence and safety. Ops needs to know how to interact with authors of change, and change authors need to know how to escalate to Ops. Examples of communication include clear escalations to on-call, only deploying changes within an author's working hours, and tools that make finding the "right" person simple.

EXTRA-TEAM COMMUNICATION

- ✓ You have a mechanism in place to communicate your larger goals to external stakeholders.

Similar to the challenges in intra-team communication, when you are deploying daily you need a new way of keeping the company at large informed about what you're up to and how you're performing.

This can be an exec's need to understand progress on a high-level goal. Often, when moving quickly, there's a need to understand how much work is "keeping the lights on" vs new initiatives. The information shared hasn't changed, but your ability to understand where you are in these larger goals is less binary. If you want to keep things humming, invest in your high-level information radiators.

- ✓ You have a mechanism in place to inform customer support when bug fixes or support issues have been deployed
- ✓ You have a mechanism in place to bring in documentation and marketing at the appropriate time as a larger goal reaches completion.

ISSUE TRACKING / PLANNING

- ✓ You are able to break your issue work into small batch sizes that roughly equate to one deploy. This is so your team can be on the same page about cadence and identify blockers. It also helps with your roll-ups to understand progress on longer-running goals.
- ✓ You have a mechanism in place that correlates your issues to when they were deployed.

REAL-TIME CHAT

- ✓ You have shared team channels with notifications about deploys to your different environments
- ✓ You have shared team channels where alerts or health information about your systems are easily consumed

- ✓ You have the ability to quickly spin up new rooms with relevant team members when responding to an incident

VISIBILITY

- ✓ You have a system in place that ties all of your tools together to provide a unified, deploy-based view of all your systems

KNOWLEDGE SHARING

- ✓ You have centralized documentation on how to troubleshoot common scenarios
- ✓ You have documented what conditions should cause an individual to roll back or forward a negative change to avoid making developers decide in the moment

Culture for one deploy a day

The cultural goal to support one deploy a day is to provide the entire team with the planning, freedom and safety needed to deploy daily. This is less of a shift than the shift to one deploy a week. However, you may find that your team still sees deploys as "risky" and you'll need to put in place the practices to dispel these ideas.

Another large cultural shift here is around batch size and getting comfortable shipping incomplete work. At one deploy a week you could still get away with shipping the whole thing in a big bang. However, deploying daily means shipping increments. This is where safety comes from, but it requires a change in mindset.

DEVELOPERS

Developers need to change two primary mindsets.

The first is how developers think about shipping incomplete work. Instead of exposing their changes to users and releasing it all in one big bang, they will need to think in terms of the smallest shippable unit. Also, when changes are risky or not yet ready for primetime, they need to think about how to ship darkly, i.e., ship the code but limit the amount of execution or exposure that code gets.

The second is changing the developers' definition of done.[15] No longer can work be considered done when a pull request has been opened and reviewed. Developers need to include deployments to multiple environments and deployment verification, plus any rework into their definition. They need to embrace being "done, done," the time when everything is exposed to users, verified, documented, shared, and you can truly move onto the next thing.

Mechanisms to achieve:

- ✓ One of the best mechanisms for being able to ship incomplete work is adopting some form of feature flagging.[16]
- ✓ It can also be highly effective to deploy every pull request one at a time. Adopting this pattern allows you to include reviewing the batch size of a change along with your pull request code review. If a batch is too large, other team members can ask a developer to break their change into multiple pull requests.

To help developers get to "done, done" you want to make sure they:

- ✓ Are empowered to deploy their own changes to all the environments you maintain and can do so quickly and without failure

- ✓ Have enough metrics and an understanding of their norms to be able to verify a deploy
- ✓ Understand how to escalate and roll back if need be

PRODUCT MANAGERS AND DESIGNERS

At this phase, PM and Design have an amazing opportunity if they are willing to embrace it. They can truly be agile, iterating in what is basically real-time with their developers. When this works, it can turbo charge the amount of value you ship to customers. Imagine being able to ship a change, test it against 50% of your users, gather in-product feedback and ship interactions to that work, all within a day or two!

PMs and designers also need to have strong trust with their dev partners so they can really understand how a project is progressing its way through to completion. Because there will be so much incomplete work shipped at any given time, PMs and designers need processes in place that they and devs have agreed to follow so everyone can get their jobs done.

Mechanisms to achieve:

- ✓ Have a deployment tracker that allows a developer to tag a pull request for PM or Design's attention in a specific environment. A tool like this will then notify, in Slack or via email, the individuals when the work has shipped. This allows real-time communication between the involved parties. Think of this as continuous demos.
- ✓ Have an agreed upon mechanism for feedback and rework, such as an issue assigned directly to a developer. One strategy for solving the incomplete work issue is for a team to adopt six-week cycles.[17] A cycle has a primary and secondary goal, and because it's short, causes not only Dev, but PM and Design to think in smaller, shippable units.

- ✓ At this phase it can also be helpful to expose user-facing feature flags to PMs so they can control the rollout.

ENGINEERING MANAGERS

Engineering managers need to become the champion of frequent deploys. They should have an understanding of where their team is by continuously measuring their team's Accelerate (DORA) metrics. Then, they can identify bottlenecks and prioritize removing them.

Providing the team with the tools for continuous improvement is a virtuous cycle. Engineering managers have the zoomed-out view that allows them to make sure the team has the process, tools and safety nets to deploy once or more a day.

Mechanisms to achieve:

- ✓ Implement tools that help you continuously measure Accelerate metrics: Deploy Frequency (to each environment), Change Lead Time (even better if you can see where this time is being spent), Change Failure Rate (first define what failure means) and Mean Time to Recovery (MTTR).

Some of these are harder to measure than others. Tools like Sleuth make measuring these values easy and they work with the best of breed tools you're already using.

Once you have measures in place, see where your team is weakest and invest in that area. For example, if your Change Failure Rate to production is high, think about adding an approval step between your staging and production deploys. Or, if you run into many rollbacks, try locking your deploys to allow for single file deploys. If your MTTR is too

high, add in an impact tracking tool so your developers can drive MTTD (mean time to discovery) down to zero.

UPPER MANAGEMENT

Upper management needs to understand that to stay competitive in our current software world, they will need to take advantage of the speed, reliability, and agility of teams practicing frequent deploys and DevOps. Without this, their organizations run the risk of disruption. Also, as the industry moves this way and talent has experienced this way of working, it's harder and harder to convince the best people to work any other way.

The importance here is in helping your entire organization to shift, not just your Dev teams. If your sales and marketing teams aren't aligned with the change, you'll have internal friction that can do irreparable damage to the organization.

Mechanisms to achieve:

- ✓ Take the time to explain to your organization how you are delivering value to your customers. Explain how incremental delivery works, what things will be released in a big bang, and how this way of working can allow you to deliver value to customers faster.

Places where organizations struggle when practicing frequent deploys are often:

- **Sales and Support:** they struggle to keep up with small changes to the product, so they have difficulty keeping customers happy. This can be combated by adopting tools that disseminate information about product changes to your organization.
- **Marketing:** they can feel like their biggest lever, the new feature launch, has been taken away from them. This can be combated with coordinating feature flag launches with marketing launches.

Who owns your one deploy a day?

As discussed throughout this chapter, getting to a deploy a day requires a lot of supporting tools and processes. Once you have these in place, there really is no good reason to keep your developers from deploying the code they've created.

It is possible to have a dedicated team or individual who does deploys at this frequency. However, you will find that it slows down other parts of your process and erodes the trust that needs to be the bedrock of your new development process.

Can the owner of a deploy a day be someone other than the developers who authored the change? Yes. Should it be? No.

There are some hybrid options here. And remember, there are many ways to get to your goal.

If empowering developers to deploy to production is blocked for a good reason, then so be it. In that case, consider empowering developers to deploy to your alternate environments and then having a gated process to deploy changes from those environments to production. You can also add Slack-based approvals for promoting deploys from your non-production environments to production.

Do all that you can to give your developers the end-to-end experience so they can own their changes and verify them in production. Only once you have this culture in place can you reap the benefits of quick MTTD and rapid improvements.

04

Deploy one hundred times a day

STORY FROM THE TRENCHES

A need so bad we had to build, and it's worth millions

I have never worked at a company the size of Facebook or Google, but I have seen first-hand how a company like Atlassian, at 4,000 employees, has approached deployment at scale. I've also done multiple interviews with leaders and engineers inside of the mega-scale organizations.

Their common experience is best summed up by a quote from the head of engineering at Shopify, "We've built all this internally by necessity: visibility into deployments, canaries, Slack integrations, dashboards, etc. This is kinda a must have for a cloud native platform and 2k engineers :)."

I've heard similar statements and sentiments from leaders at Facebook, Google, Stripe, and many others. At a point of scale each of these organizations has dedicated entire platform teams and hundreds of millions of dollars into their deployment infrastructure. As one senior manager at Facebook said, "Deployments are so well oiled at this point, even as an infrastructure team we don't even think about deploys. What we're doing at Facebook is 20 years ahead of what they're doing at Atlassian."

It's so common for a large organization to build internal tooling that some of the most popular open source deployment projects are spawned from these. Spinnaker grew out of Netflix, Kubernetes from Google, Backstage from Spotify, Clutch from Lyft, and there are many more examples.

The takeaway from this is, the larger you grow and the faster you ship the more you need supporting tooling to make sense of it all. Not every organization can afford to spend hundreds of millions of dollars on deployment tooling. And you shouldn't have to! There's a new breed of tooling out there, like Sleuth, that is bringing tools the big players have to all of us.

If you've made it this far, I have a confession to make. Deploying 100 times a day is only really something that you'll experience if you are at a very large organization working off a monolithic codebase or a very large monorepo.[18]

The more common pattern at larger organizations is to split up code bases by product or service and have each of the teams responsible for that code deploy their changes up to 10 times a day. In these cases, the organization as a whole is deploying hundreds of times a day, but by breaking things down to smaller teams they manage to avoid the coordination overhead of working off a gigantic codebase.

That said, some of the largest organizations out there, like Facebook, Google, Shopify, and others, maintain huge monorepos and have an incredibly modern deployment process.

The keys to being able to deploy a hundred times a day are:

- ✓ Highly sophisticated, custom built or modern, deploy-first tooling that makes deploys a non-event 99.99% of the time
- ✓ Canary deployments that allow changes to be rolled out to small percentages of live traffic, with highly sophisticated and automated observability that can automatically stop and roll back deploys
- ✓ An engineering department dedicated to deployments, deployment tooling, and continuous improvement of deployment environments
- ✓ Full buy-in up and down the organization that frequent deploys for shipping functionality is an integral part of how it functions
- ✓ Individual developers are provided with a powerful toolset that allows full visibility into the deployment pipeline, deployment health, and the ability to easily create new deployment pipelines based off of the organization's standards

Measurements for one hundred deploys a day

In a large organization, many of the issues you and your teams will face are issues of scale and the challenges that come from trying to have a shared language and set of measures across dissimilar teams.

Measuring Accelerate metrics in these environments is very important as they can be the drivers for spotting where, organizationally, you need to allocate resources to improve.

It's often the case at these large organizations that custom tooling has been built to provide this data. However, what's more common is that you find five to 10 different ways of measuring, each with a slightly varied definition for each metric. The challenges we've seen are in aligning internal teams on a single definition for a measure. For example, it can be a challenge to support the sheer variety of Change Failure definitions you find in a large organization.

Adopting a modern, deploy-first tool for these measures has the benefit of a common definition and a way to apply them uniformly to dissimilar teams. Your tooling should be able to report, at a minimum:

- ✓ How teams are performing in relation to each other, slice and dice those teams by programming language, reporting lines and more
- ✓ Quick access to organization-wide anomalies
- ✓ Ability to pull high-level aggregated reports about overall organization trends and performance

The tooling your teams can employ to improve here can be very custom to the organization. However, some common tools include:

- ✓ Deploy workflow automation: an automation engine that allows custom deploy workflows that can automate integrations with other important systems in the organization
- ✓ Automated high-level deploy notifications tailored for units of your organization: Support, Product Management, Sales, Marketing, Executive Management, etc.

Development practices for one hundred deploys a day

For organizations to support 2,000 or more engineers continuously working on and shipping code, they need to have become masters of all the practices we've discussed thus far. Because each of these organizations has unique needs that evolved over years, there are no hard and fast limits they need to make sure they hit. However, there are some common practices organizations of this size rely on to keep the code flowing.

RELEASE TRAINS AND FEATURE FLAGS

As discussed earlier, if your team can accommodate it, deploying each small batch of change by itself is an ideal way to reduce risk and keep the code flowing. However, if your deploy process takes 30 minutes, a very respectable deploy time, and your team is working eight-hour days, that's a maximum of 16 deployments in a day.

Now imagine your team consists of 500 developers, all shipping small batched changes. You won't be able to deploy each change by itself. The way many teams get around this are release trains.

Release trains are scheduled deploys that "leave the station" at regular and frequent intervals. In this type of deployment environment, developers know that they can merge their changes and that it will be deployed via the next train. This kind of structure is best supported when used in conjunction with feature flags. Feature flags allow code to ship darkly and be enabled separate from the deployment of the supporting code.

Release trains do have the disadvantage of deploying a larger amount of changes all in one go. That's why they are often coupled with canaries, multiple pre-production environments with approvals, smoke tests, and automated rollbacks, each of which we discuss next.

CANARY DEPLOYS

When deploying hundreds of times a day to systems at scale, the reliability of your changes are paramount. Canary deploys are when you deploy your changes to a subset of your infrastructure. Once happy with the health of those changes you start to flow a small percentage of your user traffic through that infrastructure. If, through automated observability, that canary traffic is deemed healthy you increase the percentage of traffic flowing through your new changes. If, at any point, an anomaly or unintended error is detected you revert that user traffic to your older version. In this way you can reduce the blast area of any negative changes.

Canary deploys are exceptionally powerful; however, they do come with a price. It can require a much larger investment in infrastructure and support for that infrastructure to maintain a canary setup (although

modern tools like AWS often support this with little effort if you follow their best practices). Another main cost is time. Because you are allowing your changes to soak in your canaries, you will have a longer total time to deploy.

Another cost is that it becomes harder to understand when a change has fully made its way into production. Teams will need to invest in some form of tooling that makes it easy for developers to understand when their changes have fully rolled out in production.

Finally, teams need to invest heavily in automated health detection to create and maintain reliable and automated measures that can determine if a canary environment needs a rollback.

PRE-PRODUCTION ENVIRONMENTS WITH APPROVALS

Another way teams at scale bake in reliability are through multiple pre-production environments. Things like a QA environment provide external testing teams with an integrated place to run their tests. A staging environment is a great place for developers and PMs to spot check changes before they hit customers.

Pre-production environments also allow teams to execute automated smoke tests, tests that simulate end user behavior as if your product is a black box. Smoke tests can be used as an automated gate to environments that are further down your deployment pipeline.

One powerful tool that larger and smaller teams employ are manual approvals for promotion of a change from one environment to another. As deployment workflows get more complex and changes flow through more and more pre-production environments, it can be hard to know when changes are about to deploy to production.

Putting in place an approval process where authors of change need to "give a thumbs up" before their changes ship can provide teams the opportunity to take a breath, perform manual verifications and other tasks before their changes deploy to production.

Most modern teams use a streamlined process where approvals are automated and performed in a communication tool like Slack. Tools like Sleuth and Harness provide these deployment tools out-of-the-box for teams.

Communications for one hundred deploys a day

At the scale of a hundred deploys a day, there are so many moving parts and participants that any breakdown in communication lines will grind the whole process to a halt.

Teams working at this scale have invested heavily not only in the practices and tooling described above, but also in the communication lines associated with those tools and processes. To support 2,000 or more engineers continuously working on and shipping code, they need to have become masters of all the communication lines and tools we've discussed thus far.

The goal at this scale is to make sure:

- ✓ The process for how and when to deploy and roll back are fully automated and self-explanatory. Individuals must be able to easily and clearly see the state of systems and where their changes are in relation to their deployment workflow.

- ✓ Mechanisms must be in place to ensure that none of the many systems drift out of a uniform state; alarms and notifications should alert teams when drift has breached a set level.
- ✓ You need a single pane of glass to tie together all your best-in-class tools.

ADVANCED DEPLOYMENT WORKFLOW AND PIPELINE COMMUNICATIONS

In this phase, the sheer number of deployment environments and changes flowing through your systems can be overwhelming. It's critical that there is an easy, accessible way to visualize the state of your deployment pipeline and any changes moving through it.

There are any number of ways an organization can surface this information. Most organizations in this phase have invested in custom tooling to accomplish this. The common thread is to invest enough such that developers feel like deploys are non-events and have become so comfortable that they don't even question how their changes make it out to users.

Some common ways this is accomplished are:

- ✓ Well known and published "train schedules" so teams know how and when to get changes out.
- ✓ Automated systems, often via Slack, that can tell a developer where their changes are in a deployment pipeline.
- ✓ Automated processes for when things get stuck or deviate from normal, so that an author of change knows when this happens and exactly what next steps are. It's key that this kind of situation can't hold up any other deployment activities.
- ✓ Health metrics are baked into the automated deployment process so that authors of change don't need to monitor for them, they just

know that if they are breached their changes won't flow to production. They will be alerted and know what to do next.

DRIFT DETECTION

Multiple pre-production environments and canary deployments mean your team needs to know when environments or subsets of production have drifted apart, by how much and what exactly is in that difference.

Teams at the top of their game have built tooling that allows anyone to see exactly the state of their supported environments and even the ability to trigger a synchronization. At scale it becomes important to have alarms and monitors around this drift, so no two systems drift past a set threshold. Tools like Sleuth can keep track of drift across your environments.

A SINGLE PANE OF GLASS

Throughout this book we've discussed many tools and how they help your teams. However, as tools proliferate, the sheer number of things to keep track of and places to visit for information become overwhelming and dilute the potency of the tools. Teams at this scale need an aggregated, deploy-centric view of all relevant information.

The deploy is the unit that matters; it's the unit that ships to customers. As such, it's critical to know which issue, what pull request, which metrics, what errors (and more) are related to a deploy. Teams operating at this scale have often invested millions in custom solutions to link their best-of-breed tools. Often teams will try to link everything together via their issue tracker, but this tends to fall down when tracking multiple deployment environments.

In the end, at scale, a deploy-based view of your software and communications and operations is the way to make sense of the fast pace of change.

Culture for one hundred deploys a day

When an organization is deploying hundreds of times a day, that organization has decided to go all in on frequent deploys for the speed, power and value it offers. It will have likely invested in a continuous engineering effort to support this culture. These organizations didn't get here overnight. They didn't go from deploying once a quarter to deploying hundreds of times a day.

If they have transformed, they did so a step at a time, moving to once a week, then once a day and then truly flexing their deployment muscles. If your organization is looking to transform, keep this in mind. You must learn how to walk before you can run!

When an organization has reached this level of sophistication internally, it feels like "things have always been done this way" and that working any other way would be unnatural. Because it's an ingrained part of the culture, new hires don't have to be convinced, they need to be included.

An organization won't have reached this level unless there's organization-wide buy-in. The tooling and organizational investment required to pull it off is expensive – if there were skepticism, the investment would

have been withheld and the effort would have floundered. Rather than discuss the shift in cultural attitudes as we have in previous chapters, let's instead describe what the culture for each role looks and feels like.

DEVELOPERS

- ✓ Developers understand that their work goes out in small batches and are aware of the tools at their disposal, such as feature flags, that allow them to ship larger functionality incrementally.
- ✓ They have complete confidence that there's an institutional safety net that means bad changes are highly unlikely to make it to production and if they do, the blast radius is small.
- ✓ They have confidence that their team understands how things work and there's a sense that they're all in it together.
- ✓ Developers feel like deployments and safety are not the bottlenecks to their productivity.
- ✓ They also have the confidence that they can make a mistake and ship a fix "in no time."
- ✓ They have the information they need to get the job done and deploying feels like a non-event.

PRODUCT MANAGERS AND DESIGNERS

Product managers and designers are finally part of the agile process. Because they have integrated environments to test in and because changes can be shipped quickly, they can run tests and ship the results of those tests to truly satisfy the needs of their users.

They have also reduced the batch size of their work, building out smaller increments of work because they can see them realized quicker. They know how to communicate to users even though things are moving quickly.

ENGINEERING MANAGERS

Engineering managers get to be the team's champion for continuous improvement. As a manager you know that working in a DevOps way is a never-ending quest for better. This is a virtuous cycle. As you clear large roadblocks smaller ones come into focus.

An engineering manager also no longer needs to steal time for tech debt. Because reliability and quality are key ingredients to moving faster, they must be baked into the process.

Also, at the scale of a hundred deploys a day you will have external teams working on deployment tooling, metrics and visibility that will allow your team to punch above its weight.

UPPER MANAGEMENT

Upper management knows that their ability to ship continuously is a competitive advantage. They understand that it's an enabler to scale their engineering organization to dizzying heights. They understand that it provides them with the tools and mechanisms to provide the reliability customers expect, and that gives them further competitive advantage.

They've also learned that this isn't cheap. That only through continued investment can they truly achieve the goal. But they are clear on the return they receive from that investment.

Management now gets to focus on issues of scale instead of worrying about the organization's ability to ship.

CUSTOMERS

Customers are almost always the largest beneficiaries of an organization's ability to ship continuously. They see tangible benefits, such as quick product evolution, quick resolution of bugs, better reliability and scalability, and quick response to feedback.

The industry in general is still struggling to figure out how to make rapid changes without, at times, confusing the customer. If a button was in one place one day, another a week later, and then a third place a month after that, it can be a jarring experience for customers. Support teams and customers are still trying to balance how best to absorb the pace of change organizations are now capable of making.

Who owns your one hundred deploys a day?

Whether an organization is achieving a hundred deploys a day via many small teams deploying 10 times a day, or by utilizing release trains and segregated deploys, they have one thing in common: they need to have dedicated engineering resources to work on deployment pipelines, tooling and platform.

The goal at this scale is still to empower developers to get their changes out to production under their own steam and quickly, reliably and safely. However, having many independent teams all building their own unique tools poses challenges.

Organizations at scale need to worry about things like SOC2, ISO 27001 and FedRamp certification. They'll likely have board-level security commitments to adhere to, and commitments to auditability smaller organizations don't need to worry about.

In large organizations there are always new large and small projects spinning up and down. There's developer spikes, new product proof of concepts, and many other such projects. An organization at scale needs to be able to provide their developers with a way to provision new deployments quickly and easily in a way that complies with the organization's existing obligations and objectives.

Doing this via hundreds of one-off deployment pipelines just isn't feasible. Most organizations at scale have learned this along the way and continuously invested in deployment capabilities. They have teams that provide the raw deployment ingredients for other teams to perform their frequent deploys. It's often the case that teams still configure their deployment pipelines. However, they're often utilizing and working with such centralized teams and tools to do so.

05

A continuous journey toward improvements

One key characteristic shared by elite software teams is that they have baked continuous improvement into how they work.

Elite teams are continually measuring how they are working, identifying the area most ripe for improvement, modifying their process and then repeating it all over again.

These teams have learned that frequent deploys, DevOps and the culture that comes with it are a continuous journey, not a destination.

We've shown throughout this book that moving to frequent deploys comes with a wealth of benefits for your customers, your developers, your organization, your culture and your ability to innovate. It's a journey that requires commitment and investment but one that is worth every bit of money and effort invested.

If your team is considering starting on your journey, remember that you are not alone. Many surveys have shown that we are witnessing a mass adoption of DevOps practices with almost 45% of teams surveyed saying they've been practicing for a year or less.[19]

What this means for you is that there are amazing resources for you to ensure that your team can make a successful transition to frequent deploys.

Start at the start, focus on the step right in front of you, always be measuring, and rely on the tools to help you improve. Your path to deploying a hundred times a day awaits you!

References

1. DevOps is a set of practices that combines software development and IT operations. It aims to shorten the systems development life cycle and provide continuous delivery with high software quality.
2. Continuous delivery is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time and, when releasing the software, doing so manually.
3. Forsgren, Nicole, Dr., Frazelle, Jessie, Humble, Jez, Smith, Dustin, Dr. “Accelerate: State of DevOps 2019.” DevOps Research & Assessment, Google Cloud. 2019: 22.
4. Forsgren, Nicole, Dr., Frazelle, Jessie, Humble, Jez, Smith, Dustin, Dr. “Accelerate: State of DevOps 2019.” DevOps Research & Assessment, Google Cloud. 2019: 22
5. Forsgren, Nicole, Dr., Frazelle, Humble, Jez, Kim, Gene. “Accelerate: The Science of Lean Software and DevOps” IT Revolution Press, 2018
6. A blameless culture assumes that everyone involved in a mishap has good intentions and did the right thing with the information they had.
7. CI/CD refers to the combined practices of continuous integration and either continuous delivery or continuous deployment. CI/CD bridges the gaps between development and operation activities and teams by enforcing automation in building, testing and deployment of applications.
8. Forsgren, Nicole, Dr., Frazelle, Humble, Jez, Kim, Gene. “Accelerate: The Science of Lean Software and DevOps” IT Revolution Press, 2018
9. Failure can be defined as many things, from having created an incident to causing an important metric to degrade. It’s up to you how you want to define a failure.
10. Pull requests let you tell others about changes you've pushed to a branch in a git repository. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.
11. Flakey tests are tests that don’t pass 100% of the time. This is often because of a poorly written test or because the test runs against an inconsistent external resource.
12. Service Level Objective - the objectives your team must hit to meet your SLAs - the agreement you make with your customers or end users.
13. MTTD - mean time to detect or how long it takes to discover change failure.
14. A continuous delivery pipeline is an implementation of the continuous paradigm, where automated builds, tests, and deployments are orchestrated as one release workflow.
15. The Definition of Done is an agreed-upon set of items that must be completed before a project or user story can be considered complete.
16. A feature flag is used to hide, enable or disable a feature or code path during runtime.
17. https://m.signalvnoise.com/how-we-structure-our-work-and-teams-at-basecamp/
18. A monorepo (mono repository) is a single repository that stores all of your code and assets for every project.
19. 2020 DevOps Trends Survey by Atlassian & CITE Research

ABOUT THE AUTHOR

Dylan Etkin is CEO & Co-Founder of Sleuth, the leading deployment-based metrics tracker.

As one of the first 20 employees at Atlassian, Dylan was a founding engineer and the first architect of Jira. He has led engineering for products at scale in Bitbucket and Statuspage.

He has a Master's in Computer Science from ASU. He's a bit of a space nut and has been seen climbing around inside of a life-size replica of the Mir space station in Star City, Russia.

Questions? Feedback? Email us at hello@sleuth.io

SLEUTH